Reflection in Attribute Grammars

Lucas Kramer  tedinski@cs.umn.edu  Eric Van Wyk
ekrame505@umn.edu  University of Minnesota  evw@umn.edu
University of Minnesota  Minneapolis, MN, USA  Minneapolis, MN, USA

Abstract
This paper shows how reflection on (undecorated) syntax trees used in attribute grammars can significantly reduce the amount of boiler-plate specifications that must be written. It is implemented in the Silver attribute grammar system in the form of a reflect function mapping syntax trees and other values into a generic representation and a reify function for the inverse mapping. We demonstrate its usefulness in several ways. The first is in an extension to Silver itself that simplifies writing language extensions for the AbleC extensible C specification by allowing language engineers to specify C-language syntax trees using the concrete syntax of C (with typed holes) instead of writing abstract syntax trees. Secondly, a scrap-your-boilerplate style substitution mechanism is described. The third use is in serialization and de-serialization of the interface files Silver generates to support separate compilation; a custom interface language was replaced by a generic reflection-based implementation. Finally, an experimental implementation of staged interpreters for a small staged functional language is discussed.

CCS Concepts • Software and its engineering → Translator writing systems and compiler generators.

Keywords attribute grammars, reflection, meta-programming

ACM Reference Format:

1 Introduction
Strong static type systems are lightweight, yet effective, formal methods for ensuring that run-time type errors cannot happen when executing type-correct programs, and many find the benefits of type safety outweigh the restrictions that such systems necessarily impose. Some meta-programming languages and systems enjoy the benefits of strong static typing, e.g. the Kiama [Sloane 2011] system embedded in Scala, the Java-based JastAdd [Ekman and Hedin 2007] attribute grammar system, and the Silver [Van Wyk et al. 2010] attribute grammar system used in the work presented here. The type systems in these ensure that all object-language syntax trees are well-sorted, that is, they correspond to the context free grammar defining the language.

But as many have observed, for example in the “scrap-your-boilerplate” work [Lämmel and Jones 2003], this often comes at a price. Transformations over syntax trees from syntactically rich languages are cumbersome to express recursively; for example, implementing a program transformation that rewrites \( x + 0 \) to \( x \) requires code or specifications not only for the case of addition, but for all other constructs in the language. For these other constructs the specification simply duplicates each construct with its rewritten components, leading to lots of uninteresting, cumbersome boiler-plate code. Other (non-boilerplate) challenges in language development include the construction of non-trivial syntax trees and their serialization/de-serialization to/from strings.

One approach to this problem is to use a form of reflection [Demers and Malenfant 1995]. In our approach, a (well-sorted) syntax tree is reflected into a generic form over which transformations such as the one described above can be more easily written. This ease of writing is traded for the loss of some type-safety as well-sortedness is not guaranteed when constructing or manipulating object-language trees in this form. This paper presents a technique for bringing reflection of undecorated syntax trees (those without attributes) to attribute grammars and evaluates it on several examples.

Motivating Example: The reflection system presented here is implemented in Silver and some example uses are in an extensible specification of C, AbleC [Kaminski et al. 2017a], all of which are written in Silver. Consider a language extension introducing an exponent operator to C that should translate, for example, \( x \times y \) to the code shown in Figure 1. Constructing the syntax tree of this translation by directly using abstract syntax is quite tedious and imposes barriers to entry for new language developers who must learn the numerous productions in the AbleC abstract syntax grammar to become productive. Instead it is desirable to extend the meta-language allowing abstract syntax trees to be constructed...
using the actual concrete syntax of the object-language, as
is done in systems such as Stratego [Visser 2002].

An example of this is shown on the left in Figure 2. Here
the extension developer introduces a new production (named
exponent) for an exponent operator that extends the ABLE C
host language, and specifies that exponent constructs trans-
late down (via the forwards to specification [Van Wyk et al.
2002]) to a syntax tree like the one in Figure 1. Some at-
tributes (such as $p$, the "pretty-printed" version of the con-
struct) may be defined directly on the new production, while
all other attributes (such as a translation to machine code)
are automatically computed on the forwarded-to tree.

Object-language concrete syntax, in this case C, is writ-
ten here as an expression in Silver using a "quote" pro-
duction that wraps a piece of code in the object-language
for which the abstract syntax tree should be constructed;
ableC_Expr\{...\} beginning on line 6 of Figure 2 is an exam-
ple of a quote production that introduces an Expr term from
the C language. Sometimes portions of the desired tree are
not fixed but instead should be constructed using Silver
code. Such holes in the tree may be filled in using "antique"
productions that escape from the object-language syntax
back to Silver expressions, such as $TypeExpr$ on line 7. This
line specifies the declaration and initialization of $\_res$ seen
on line 1 of Figure 1. The specification on the right of Fig-
ure 2 is what is needed when writing the abstract syntax
directly to specify this same single line. Since Silver itself is
an extensible language, language developers may easily con-
struct a version of Silver tailored to their object-language
by introducing these productions (such the ABLE C extension
to Silver used here, referred to as Silver-ABLE C.)

The reflection system presented here is used to simplify
the implementation of this extension to Silver for using
object-language concrete syntax in specifying trees. Compu-
ting the translation for quote productions such as ableC_Expr
presents a difficulty without reflection; we must generate
the abstract syntax for the Silver expression which, when
compiled and evaluated, will construct the specified object-
language syntax tree. In systems like Stratego this pro-
cess may be accomplished by the use of rewrite rules; this
is permitted as terms have a uniform/generic representa-
tion, and weak typing allows for an incremental transfor-
mation of object-language into meta-language abstract syn-
tax. However in a strongly typed system such as Silver,
object-language trees are strongly distinguished from meta-
language expressions that construct trees; thus a direct trans-
formation is required from the former to the latter.

One approach is to define a translation-to-Silver attribute
on all nonterminals of the object-language attribute grammar,
on each production writing an equation that constructs the
Silver expression for a call to that production. With nearly
500 abstract productions in the ABLE C specification doing this
would require writing a tremendous amount of boilerplate
code. An important observation is that the desired translation
does not depend on the semantics of particular productions,
but rather is only based on the name of the production and
number of children. Thus, some approach is demanded for
dealing with a generic representation of an abstract syntax
tree, rather than performing a computation on the tree itself.

There are other, similar problems that can best be ex-
pressed as generic analyses on trees. For example consider
serializing/de-serializing between a tree and a string; this can
easily be done with a uniform untyped term representation,
but not so easily with a tree represented by a variety of non-
terms. Thus we want a general-purpose mechanism that
can represent a tree in a generic way, and convert between
well-sorted trees and this generic representation.

**Reflection in Silver:** Problems of this nature benefit from
some form of reflection in the meta-language. The reflection
capability developed here consists of two primary functions:

- **reflect:** which converts a well-sorted syntax tree in
Silver into a generic tree representation of type AST.
- **reify:** which converts generic AST trees back to their
original form. This requires run-time type checking
and may fail if the generic form does not correspond to
a well-sorted object-language tree. Thus reify returns
a value of type Either<String a>: either a string error
message or the tree of the correct type (represented
by type variable a).

Transformations and analyses on AST trees are implemented
by specifying attributes and their defining equations. This
leads to specifications that are much less verbose with much
less boilerplate code.

This follows a common thread found in a wide variety
of reflective systems; one in which data can be viewed in
two ways. The first is through the regular type system of
the language, be it Java objects or Silver well-sorted syntax
trees. The second view is a more generic one in which values
of different types or sorts can be viewed in a uniform way;
this may be by reflective methods in Java available to work
on an Object super-type (Field.get. Method.invoke, etc.) or,
in our case here, though the reflect transformation into a
generic AST representation.

Figure 1. The translation of an exponent expression $x \ast\ast y$,
where $x$ is a float and $y$ is an int. Note this uses a statement-
expression (\{(\ldots; \ldots;\})\}, a C extension supported by GCC
and other compilers, to embed a statement in an expression.

```
1  (( float _res = 1;
2  for (int _i = 0; _i < y; _i++) {
3    _res \ast\ast x;
4  }
5  _res; })
```
Contributions: The primary contributions of the paper are enumerated below.

- We integrate reflection and attribute grammars using a special AST nonterminal and a bidirectional reflect/reify transformation between AST trees and well-sorted object-language syntax trees; discussing the system design in Section 3 and its implementation in Section 8.
- We demonstrate the use of reflection in mapping object-language concrete syntax to the \textsc{silver} constructs that would construct it, supporting specifications like those in Figure 2. This saved almost 18,000 lines of specification over several \textsc{ablec} extension specifications, amounting to 40% of the code base by character count. The code generation for many extensions was complex enough that they would likely not have been implemented using the direct method of specifying translation as shown on the right in Figure 2. (Section 4)
- We demonstrate the use of reflection in attribute grammars to implement “scrap your boilerplate”-style \cite{Lamme17} substitutions of particular subtrees within much larger abstract syntax trees. This saved over 2,500 lines of specification between \textsc{ablec} and several extensions, amounting to 11.8% of the \textsc{ablec} specification code base. (Section 5)
- We use reflection in conjunction with \textsc{silver}’s attribute grammar and parser specification features to provide a serialization / de-serialization library for arbitrary trees. This allowed 1,698 lines of hand-coded environment tree serialization/de-serialization code in the \textsc{silver} compiler to be replaced with only 257 lines, removing 3.75% of the \textsc{silver} code base. (Section 6)
- Use of reflection to allow a staged language evaluator to be specified as an attribute grammar and directly interpreted. (Section 7)

\textsc{silver} version 0.4.1 \cite{VanWyk10} \cite{Kaminski17} is the attribute grammar system used in this paper. The \textsc{ablec} code and extensions to it \cite{Kaminski17} are from version 0.1.3 and the \textsc{silver-ablec} code is from version 0.1.0. Specifications from each are shown in several figures below and can be found in the corresponding repository in the directory or file name given in caption in each figure. When indicating the number of lines of \textsc{silver} code or \textsc{ablec} extension code saved by using reflection it is a comparison between these and previous versions in the corresponding repositories.

2 Background: Attribute Grammars

As this paper presents an integration of reflection into attribute grammars (AGs) we provide some background on them, and their embodiment in \textsc{silver} \cite{VanWyk10} here. In essence, attribute grammars provide a “semantics of context free grammars” \cite{Knuth68} by associating/decorating nodes in a syntax tree with attributes that carry information (semantics) about that language construct. For example, in an AG for type checking a functional language, a nonterminal for expressions may be decorated with an attribute identifying the type of the expression and another providing a context that maps variable names to their types. Equations associated with productions are used to determine the values of the attributes.

More formally, an attribute grammar $AG = (G, A, O, \Gamma, \mathcal{E})$ where $G$ is a context free grammar ($NT, T, P$) with a finite set of nonterminal symbols $NT$, a finite set of terminal symbols $T$, nonterminals $A$, a set of production rules $P$, terminal symbols $O$, and a set of attribute equations $\mathcal{E}$.

---

Figure 2. An example using \textsc{silver-ablec} to implement an exponent expression in \textsc{ablec}. On the right is shown the tedious code that would be written using only plain \textsc{silver}, for just line 7 of the exponent production on the left.
T which includes basic types for integers and strings, and $P$ a set of grammar productions. Productions have the form

$$n :: NT_0 ::= n_1 :: X_1...n_k :: X_n$$

with a left-hand side nonterminal $NT_0$ and 0 or more right-hand side symbols in $NT \cup T$. These are labeled $(n_i, n_j)$ so that tree nodes can be easily referenced. In Silver, productions are labeled as concrete if they are passed to the Copper [Van Wyk and Schwerkfeger 2007] scanner and parser generator bundled with Silver. Those labeled by abstract are not; they are used to define the abstract syntax of a language and are our primary focus here.

The set of attributes $A$ is partitioned into synthesized attributes $A_S$ that propagate information up the syntax tree and inherited attributes $A_I$ that propagate information down the tree. In the example above the attribute for types is synthesized and the attribute for typing contexts is inherited. Attributes can hold various types of values beyond primitive types such as strings and integers. Vogt et al. [1989] introduced higher-order attributes in which attributes contain (yet undecorated) syntax trees. Reference [Hedin 2000] and remote [Boyland 2005] attributes that allow decorated syntax trees to be passed around as attribute values; these are sometimes thought of as references or pointers to distant nodes in the syntax tree. An occurrence-relation $O$ indicates which attributes decorate which nonterminals and a typing context $T$ tracks the types of attributes and productions for use in type checking the equations that are used to determine the values of attributes for a specific syntax tree, perhaps as produced by a parser for the language.

Equations in $E$ are associated with a single production $p \in P$ and typically have the form $n_i.a = e$ with $n_i$ being a label for a symbol in $p$. If $i = 0$ then $a$ is a synthesized attribute used to compute a value $e$ with $n_i$ otherwise $a$ is an inherited attribute computing a value for a child of $n_0$. Equations can be written separately from their production definition in aspect productions.

### 3 Design of the Reflection System

In this section we describe the design of reflection features and how they are used in attribute grammars. The implementation is discussed later in Section 8.

We first introduce the AST type, a generic representation for well-sorted syntax trees. There are two operations over these: reflect which transforms a well-sorted syntax tree into a generic AST, and reify, which maps generic trees back into well-sorted trees.

Part of the reflection system is a Silver library that defines AST as a nonterminal, shown in Figure 3 on line 2. AST trees are constructed in the same way as other trees in Silver through the application of productions to primitive values or other trees. Each AST production in Figure 3 corresponds to a type (or class of types) in the Silver language: e.g.

```
1 grammar silver:reflection;
2 nonterminal AST;
3 abstract production nonterminalAST
4 top::AST ::= prodName::String children::ASTs
5 abstract production terminalAST
6 top::AST ::= terminalName::String lexeme::String
7 location::Location
8 abstract production listAST
9 top::AST ::= vals::ASTs
10 abstract production stringAST
11 top::AST ::= s::String
12 abstract production integerAST
13 top::AST ::= i::Integer
14 abstract production anyAST
15 top::AST ::= x::a
16 nonterminal ASTs;
17 abstract production consAST
18 top::ASTs ::= h::AST t::ASTs
19 abstract production nilAST
20 top::ASTs ::= 
```

Figure 3. Some of the Silver nonterminals and productions used to represent abstract syntax trees. Production bodies, which contain attribute equations, are not shown. See core/reflect/AST.sv

stringAST to String, nonterminalAST to nonterminals (well-sorted trees), terminalAST to terminals (tokens returned from a scanner), listAST to Silver lists. Trees built by the polymorphic anyAST production contain values (represented by type variable $a$) that do not typically occur in abstract or concrete syntax trees and for which we cannot inspect in the same way: this includes productions, functions, and references to decorated trees. The ASTs nonterminal encodes a sequence of AST trees.

The reflect operation uses these productions to produce an AST value from any value and thus has the type $AST ::= a$. (A more familiar writing of this type might be $a \rightarrow AST$, but functions use the same type $::=$ notation as productions.) For example, reflect will transform the Silver tree

```
integerConstant("1", false, noIntSuffix())
```

on lines 11-13 on the right of Figure 2 into the AST tree

```
nonterminalAST("integerConstant",
consAST(stringAST("1"),
  consAST(booleanAST(false),
    consAST(nonterminalAST("noIntSuffix", nilAST()), nilAST())))
```

As with all nonterminals, attributes may be associated with the AST nonterminal. Aspect productions then associate new equations for the abstract productions in Figure 3. Most
uses of the reflection system do this. The Silver-ableC extension discussed in Section 1 introduces a new translation attribute on AST and provides equations for this attribute on AST production to produce the Silver abstract syntax that is needed, as described in more detail in Section 4. Another aspect-defined attribute is serialize, used to print out an AST tree and discussed in Section 6.

In many of the applications discussed in the following sections a generic AST tree is converted back into a well-sorted object-language tree by the reify operator. This has type Either<String> ::= AST and takes an AST and returns either a String error message or the well-sorted tree. Because AST trees can be constructed directly using the productions in Figure 3 it is possible to construct trees that do not correspond to a well-sorted tree. An attempt to reify such a tree will result in an error, returning left(s), where s is an error message of type String. A successful reification wraps the tree of type a in the right constructor. The reflection system satisfies the following invariant for any tree t:

\[ \text{reify}(\text{reflect}(t)) = \text{right}(t) \]

In practice large AST trees are rarely constructed directly; they usually are constructed by \text{reflect}. Thus ill-sorted ASTs (and thus failures when calling \text{reify}) rarely occur. Even when an error does occur it is reported immediately by the call to \text{reify}, and we have found finding and fixing the source of such errors to be straightforward.

Despite this, during the reification process we must check that the AST in question actually corresponds to the inferred result type of \text{reify}. This requires a run-time type checking process that looks up productions and terminals to ensure they are defined and match the expected types. Since Silver supports polymorphic algebraic data types (nonterminals such as Either<a b>), it may not be possible to compute the final type of a sub-tree from only its arguments, so type checking also requires full Hindley-Milner type inference.

4 Reflection for Tree Construction Using Object-Language Concrete Syntax

This section further describes the reflection system in Silver and demonstrates its use in an extension to Silver itself that lets language developers specifying trees in the object-language being developed using the concrete syntax of that language instead of its inconvenient abstract syntax. We demonstrate this on ableC, an extensible specification of C, defined in Silver. The extension to Silver for ableC is referred to as Silver-ableC below. Since C is a large and complex language the ability to write concrete syntax instead of abstract saves a tremendous amount of effort on the part of the language developer, and has lead to the development of ableC extensions that we may not have even undertaken without this capability. This was briefly demonstrated in Section 1 by Figure 2; the question we answer here is how reflection is used in translating the nice specification in the left of Figure 2 into the implementation on the right.

The step-by-step example use of Silver-ableC is shown in Figure 4. At the top left is a simple term written using Silver-ableC, while beneath it is an equivalent piece of code written using plain Silver. The goal of the Silver-ableC extension is to translate the Silver-ableC code into the same abstract syntax tree as would result from parsing the plain Silver code. To keep things manageable, a simpler example than in Figure 2 is used to explain this process and the Silver-ableC extension. The antiquoted expression \( e_1 \) is written in bold and we track its translation through both processes of using and not using Silver-ableC.

The name \( e_1 \) will hold an ableC tree that is to be plugged into the expression to be added to the C variable \( x \) and divided by 2. This is a shorter example, but qualitatively the same as on the left of Figure 2. Below this is the equivalent Silver source specification that does not use the Silver-ableC extension and requires one to write out the abstract syntax explicitly. This is a shorter example of the same thing as shown on the right of Figure 2. Both of these examples are functionally equivalent specifications and both need to translated into the same thing — a Silver abstract syntax tree.

In the plain Silver specification we see that division is represented by the ableC production ableC:divExpr (line 1) and the name \( e_1 \) (line 3) is the first argument to the addition production ableC:addExpr (line 2). This is simply parsed by the Silver compiler to generate the abstract syntax representation in the lower left of Figure 4. Since Silver is bootstrapped in Silver the abstract syntax of a Silver specification implemented as a Silver attribute grammar. Thus the resulting abstract syntax is mostly applications of various Silver productions; the production silver:applyExpr represents the application of productions (tree creation). We see a few instances of silver:applyExpr production with the first argument being the name of the production in the object-language, in this case ableC, and thus the names as strings are those fully qualified names from the ableC grammar and thus the string "ableC:divExpr" is seen on line 2 as this use of applyExpr represents the application of this ableC production for division. The second applyExpr is the list (constructed by silver:consExpr and silver:nilExpr productions) of trees to become the children of the constructed

\[ ^3 \text{In Silver, names of productions and attributes have fully-quantified forms that include the name of the grammar in which they were defined, a bit like fully qualified names in Java (but using colons instead of dots to separate names). Thus ableC:addExpr is the addExpr production in the ableC grammar. For reasons of brevity, we use ableC as the grammar name here but the actual grammar name is edu:umn:cs:melt:ableC:abstractsyntax:host, as can be seen in the specifications in the repository. Finally, note that when the context permits we will also refer to grammar elements using their shorter un-qualified name.} \]
### Figure 4. The translation process for an ableC quote production. In bold is shown an antiquoted piece of Silver code; note that it is the same between the original parse result (top right) and the final translation abstract syntax tree (bottom left). Production and grammar names have been shortened for clarity.

The first step is again parsing, but now using the Silver-ableC parser. This yields a Silver tree containing quote (line 1) and antiquote (line 4) productions in the upper right box of Figure 4. The quoteExpr on line 1 contains the ableC abstract syntax that was written out directly in the plain Silver specification. The Silver-ableC parser is a combination of the concrete syntax for Silver and ableC with syntax for the quote and antiquote productions for different ableC nonterminals (such as ableC:Expr). The antiquote production, parsed from $Expr{e1}$, switches back into Silver abstract syntax. Thus line 5 here matches line 7 in the tree in the lower left.

Using reflection: We now consider the other path to this abstract syntax tree, using C concrete syntax in the Silver-ableC extension and the Silver reflection system. The first

tree representing division: the first element being the representation of the addition, the second the constant 2. We see on line 7 then name e1 used as a Silver variable reference form of expression. From an abstract syntax tree like this, the Silver compiler will type check the specification and generate the Java code that forms the attribute grammar evaluator [Van Wyk et al. 2010].
A few of the quote and antiquote productions introduced by Silver-ableC are shown in Figure 5. Quote productions such as quoteExpr, being extensions to the Silver language, must specify the equivalent silver:Expr that they translate down to. This is done via forwarding; the details of which are not important for understanding reflection in Silver and thus reading "forwards to" as “translates to” is appropriate.\(^6\) This translation is the result of applying reflect to the syntax tree e,\(^7\) converting the ableC:Expr into a tree of type AST. These can be seen in the lower right of Figure 4; all productions (including antiquote ones) have been expanded into their nonterminalAST counterparts: ableC:divExpr on line 2 becomes nonterminalAST(“ableC:divExpr”...).

From the reflected AST, we access the translation attribute (written .translation), seen on line 7 in the middle-right and line 17 in the lower-right of the figure. This attribute constructs the Silver abstract syntax tree in the lower left, where ableC productions have been converted into silver:Expr trees build by applyExpr as discussed above. The same is true for the contents for the antiquote productions which have been reified directly into silver:Exprs. Note how the antiquoted Silver tree in the initial Silver-ableC

abstract syntax, shown in bold, has been preserved unchanged in the final abstract syntax tree.

Some equations for translation are shown in Figure 6. Every kind of Silver value represented by AST has a straightforward transformation into a piece of Silver code, for example an integer value (integerAST on line 7) is translated to a Silver integer: intExpr(i) on line 9. Here an aspect production is defining the translation attribute on a production defined in the silver:reflection grammar (line 14 of Figure 3). An attempt to translate back to Silver an anyAST tree (line 9) results in an error being raised as these are non-inspectable values that can be reified, but little else.

However, there is a complication introduced when dealing with antiquote productions: they contain a piece of Silver code that should be evaluated to obtain a tree and thus their translation should just be the wrapped piece of code. Unfortunately, this silver:Expr tree is no longer directly available as it has been “accidentally” reflected and turned into an AST. So when an antiquote production is encountered by the translation attribute, the child AST must be reified back into the silver:Expr that was originally specified.

When translating a nonterminalAST, some method of identifying whether a production is an antiquote production is required. This is done by way of a collection production attribute (line 17). The attribute antiquoteProds will contain the full names of all known antiquote productions. Extensions, such as the Silver-ableC extension which imports this grammar, can add the names of its antiquote productions to this attribute. This is done using another aspect production on line 14 in Figure 5 using the <- operator for injecting new elements (names) into this list; these are combined using the list append <> operator as specified on line 18 in Figure 6. Using this, the name of a production being translated may be looked up, as can be seen in on line 21. If the name is not in the list, the AST is translated normally, otherwise it is reified for evaluation. Thus, the AST translation code forms a library for use by tree-literal Silver extensions that does not itself have any special handling for particular object-languages.

The use of antiquoting and object-language concrete syntax has been extended beyond the core implementation described above. Since many languages have notions of lists in their grammar, for example a sequence of function arguments, the Silver-ableC extension allows one to write list-like expressions in such a sequence. For example, in

\[
ableC_{\_Expr} \{ \text{foo}(\$Exprs(a), 3, x) \}
\]

the list of expressions a, represented by the Exprs nonterminal, is naturally incorporated into this enclosing list of arguments. Another useful extension lets the concrete syntax of the object-language be used in writing patterns. (Silver does pattern matching on trees much the same way that languages such as ML or Haskell do on datatypes.) Note that these useful features add some complexity to the implementation that is not shown here.

---

\(^6\)Antiquequote productions are effectively extensions to the object-language. To satisfy the requirements of the modular well-definedness analysis [Kaminski and Van Wyk 2012] they should forward to a translation in C. However, there is no semantic equivalent in the object-language and these will never have attributes accessed on them. Thus they forward (not shown) to a dummy value that raises an error if it is erroneously evaluated.

\(^7\)The use of new(e) ensures the undecorated version of e is used; trees in Silver are treated as decorated when either is valid, which reflect would (undesirably) return wrapped in anyAST.
Figure 6. Some of the equations in the Silver library for computing the translation of AST to silver:Expr. See silver/hostEmbedding/Translation.sv

Discussion: Although this discussion has focused on Silver-ABLEC, any object-language implemented in Silver can easily benefit from the ability to write concrete syntax for constructing trees and patterns. Adapting this for another object-language amounts to specifying the concrete and abstract (Figure 5) syntax for quote and antiquote productions for the language, then importing this grammar and the concrete syntax of the object-language into another Silver instance.

In a complex language such as C, the use of concrete syntax over abstract syntax is a tremendous savior of time and effort. Trees can be specified directly and any syntax errors in the object-language concrete syntax are detected when the language specification is compiled, with error messages pointing to the precise location in the Silver specification8. We implemented several extensions to ABLEC using this new approach, including an embedding of Prolog and a supporting unification framework. These involve generating quite a bit of plain C code and we may not have attempted these without being able to use of the Silver-ABLEC extension to Silver. We did modify the implementation to count the number of characters that are saved by writing in the object-language concrete syntax (the left side of Figure 2) instead of the abstract (the right side). For all extensions that use Silver-ABLEC this saving is over 775K characters (or almost 18K lines of code based on pretty-printing the generated code with a maximum line length of 80). This is about 40% of the would-be code base size of 1.92M characters. Note that this savings came from writing new extensions that used this feature from the beginning, not from removing and replacing specifications with the concrete syntax extension.

5 Reflection for Substitutions on Trees
A common problem with processing complex tree-structured data is to update or substitute the value of a particular subtree, without introducing large amounts of boilerplate. This sort of problem may arise in a compiler when implementing a feature such as C++ templates, where we wish to replace all occurrences of a name with the tree for a particular type or expression. An example of this transformation (shown in Figure 7) comes from ABLEC in which a C++-style template extension (top) is instantiated to plain C code (bottom) by substitution.

Figure 7. Instantiating an ABLEC C++-style template extension (top) into plain C code (bottom) by substitution.

8 The source location of a tree in Silver is represented as an annotation. These are extra pieces of information attached to syntax tree nodes, but are omitted here for clarity as they are handled essentially the same as children. Thus location information is preserved in a reflected AST.
Figure 8. A portion of the ableC substitution library. See
edu.umn.cs.melt.ableC/abstractsyntax/substitution/Substituted.sv
to reify leads to an error (line 23) but otherwise the resulting
Decl tree d is extracted from the Either value from reify.

These attributes decorate AST trees, for which aspect pro-
ductions are used to define the substitution behavior. On all productions aside from nonterminalAST, such as stringAST
on line 14, substituted consists of the same production but with children that have been substituted, as it only
makes sense to define substitutions for particular productions and not strings or lists. The substituted equation on
nonterminalAST (line 9) first maps the substitutions in top.subs
over itself (top) and foldl folds up the Maybe results using
fromMaybe to pick the first successful substitution (which
would also include substitutions applied to its children). If there are none it returns the tree with just the substituted
children (line 10). The function typeExprSubstitution on line 26 returns a substitution (Sub) given a string n to replace with a type expression sub. It returns a lambda function taking
an AST that reifies its input and, if successful, matches it
against a pattern (line 29) to possibly return a different AST. Here if a is type expression for a typedef name it checks if the names are the same (line 30) and if so returns the substitution (sub)
wrapped in just to indicate success.

The original implementation of the ableC substitution
library defined a similar substituted equation for every pro-
duction in the ableC abstract syntax. These all follow the
same pattern and were replaced by this substitution grammar,
replacing over 2,500 lines of boilerplate code in the ableC
specification (11.8% of the code base), in addition to numer-
ous similar equations on ableC extension productions.

6 Reflection for Automatic Tree
Serialization and De-serialization

A significant source of boilerplate specification in the Sil-
ver specification of the Silver language is in its module for
serializing and de-serializing syntax trees that represent the
environment for a grammar. These environments indicate
what grammar elements, such as attributes, productions, and
nonterminals, are declared and certain properties of them,
such as their type. The Silver compiler separately compiles
grammar modules and uses the serialization of environments
as interface files to avoid reading an unchanged source gram-
mar imported by another grammar being compiled. The orig-
inal implementation of serialization of environments was
done by defining an attribute on all nonterminals and their
productions, representing the environment as a string, and
de-serialization by a parser using a custom grammar.

Using reflection and AST trees we can define a generic
implementation of this, replacing 1,698 lines in the Silver
compiler with 257 lines, and saving 1,441 out of 38,400 lines
(3.75%) of the entire code base. This change does come at a
cost, as the generic interface files are on average 215% larger
than before (measured for the ableC code base), less readable
by humans, and adds 2-3 seconds to builds taking around 60
Figure 9. An example of serialization/de-serialization.

seconds. However we believe such penalties are worth the significant savings in code complexity.

A visualization of the serialization/de-serialization process is shown in Figure 9. At the top is a simplified notion of an environment consisting of a list of two pairs, mapping x to 4 and y to 2. Serialization is done by reflecting a tree into an AST, the middle box of the figure, and accessing the serialize attribute on the AST, which then produces the string value at the bottom. This attribute, lines 1-3 of Figure 10, is actually of type Either<String String>. The reason for this is that serialization may fail if a non-printable value, such as a function, is a component of the reflected tree constructed as a anyAST tree (line 20). The left side of Either encodes an error message and the right side the successful result, as seen before in the return type of reify. Integer and string ASTs (lines 12 and 16) are serialized as expected, with string special characters needing to be escaped first.

Of interest is the serialization of nonterminalAST trees (line 8). This expression uses a monadic computation using Haskell-style do notation. In Silver the bind and return operation are not inferred as Silver does not have type classes and must instead be specified explicitly. Here cSer is the result of serializing the child nodes. If any of these fail the monadic computation passes the failure message along automatically, otherwise a string using this, the production name, and appropriate parens is constructed.

For de-serialization, we use Silver’s built-in declarative parser specification features to define concrete syntax matching the serialization strings. The generated parser constructs AST trees that are then reified to recreate an environment, as illustrated on the right of Figure 9. Together, these provide a concise and convenient mechanism for specifying serialization and de-serialization of Silver syntax trees. 

7 Reflection for Implementing Evaluators for Staged Languages

In this section we provide the final example of reflection in attribute grammars to implement an evaluator for a simple staged programming language, a subset of MetaOCaml [Kiselyov 2018]. Staged programming is a programming paradigm where pieces of code may be constructed at runtime, passed around as values, and eventually run, all in a type-safe
way. Examples of such languages include MetaML [Sheard 1998] and MetaOCaml. This paradigm can provide performance improvements by generating a specialized piece of code that is used more than once. The canonical example in this area is a staged power function to efficiently compute an exponent \( a^n \) by generating code specialized for a given value of \( n \); a MetaOCaml implementation of this is shown in Figure 11. Such languages use strong typing to allow well-defined semantics when running generated code [Filinski 1999]. Calcagno et al. [2003] describes an approach to implementing staged languages by translation to a lower-level language containing constructs for reflection and term construction. One can also build a direct evaluator for a staged language, the approach we have chosen to demonstrate.

In addition to all the standard functional programming constructs, MetaOCaml has 3 new operators: quote (\(.>\).), escape (\(-\)) and run (\(\!.\)). Quote constructs a value of type ‘a code, where ‘a is the type of the quoted expression, corresponding to a fragment of code being constructed. The escape operator can be written inside of a quoted expression, indicating that the escaped expression, when evaluated, will yield a piece of code that should be plugged into the result. The run operator executes a code value. Static typing ensures that no run-time type errors occur in evaluating quoted code, although run-time checking is required to ensure that some corner cases involving free variables are not evaluated (e.g. \(<\text{fun} x -> \text{.}(\text{.}! \text{.<}x\text{.}>)\text{.}, which is still well-typed).

The reflection system provides what is needed to elegantly implement our subset of MetaOCaml using attribute grammars. The interpreter is structured using an Expr nonterminal in the language’s abstract syntax. In addition to the usual productions for expressions, there are productions quote, escape and run, each of which wraps a single expression. A Value nonterminal has productions representing integers (intValue :: (Value ::= Integer)), functions (closureValue :: (Value ::= String Expr Env)), and code (codeValue :: (Value ::= AST)), among other values. An inherited attribute on Expr passes down the value environment, while a synthesized attribute computes the resulting value.

To evaluate a quote expression, the wrapped Expr is reflected, yielding an AST that contains reflected escape productions. A synthesized attribute on AST is used to find each escape production, reify the wrapped expression, evaluate the expression (using the value environment also passed down as an inherited attribute on AST) and replace the escape production AST with the AST from the resulting codeValue production. This process is somewhat similar to the previously described process of translating object-language literals or performing substitutions. Finally to evaluate a run expression, the operand is evaluated and the AST is extracted from the resulting codeValue, reified and itself evaluated.

8 Implementing the Reflection System

This section provides a brief discussion of the implementation of the Silver reflection system. Silver is implemented by translation to Java. Basic types (such as strings and integers) all have concrete Java equivalents. The Java abstract class Node represents all nonterminals; every nonterminal type becomes an abstract subclass of Node, and every production of a nonterminal is a concrete subclass of that nonterminal’s class. In this way, the Java type system encodes the syntax of the object-language being defined in Silver.

Node contains a number of abstract methods, such as to get the Silver name of a production or iterate over its children. Since no extra type-level information is required, reflect is a Silver foreign function that calls a recursive Java function to walk a Node and call the appropriate constructors corresponding to AST productions.

The implementation of reify is significantly more complex due to its run-time type-dependent nature; reify is thus a language construct in Silver (as opposed to being a foreign function as in the case of reflect). reify does run-time checking in constructing a Silver tree to ensure that it is well-sorted. To do this, reify requires the run-time representation of the type of the tree it is to produce; this type is provided by Silver’s type inference system. For AST trees representing integers or strings the reification into a Silver value of type Integer or String is straightforward. For a nonterminalAST-constructed tree, reify gets the production name (the first child of production nonterminalAST) and uses Java reflection to see if that production exists, and if so, gets the Java Class object implementing that Silver production. On this it calls a static reify method (generated as a part of the class), parameterized by the expected return type it is to construct and the array of child AST trees yet to be reified. It checks that the appropriate number of children were provided, unifies the given expected type with the actual production left-hand-side nonterminal type, then reifies the children using the corresponding right-hand-side types. Finally, it constructs and returns a new production object with the results.

To support this, runtime type information, as an object of a new Java TypeRep class, is stored on each tree node. Because Silver supports parametric polymorphism, a runtime type unification process was also added, to mirror the compile-time type unification process. This is used when checking that types are compatible when constructing new trees. If not, reify returns an error. All Silver value classes implement a TypeRep interface with a getType function returning a TypeRep value. Any type, e.g. Silver functions, or other new foreign type to be reflected into an AST tree, using the type-parametric anyAST production must implement this interface.

\[\text{Available at http://melt.cs.umn.edu/ and https://github.com/melt-umn/meta-ocaml-lite, archived at https://doi.org/10.13020/z10a-7g60.}\]
9 Related Work

Reflection, first introduced by Smith [1982], has been discussed in multiple contexts in computer science. Demers and Malenfant [1995] more precisely defines reflection, and distinguishes between structural reflection where the language provides introspection of the program being executed and its abstract data types, and behavioral reflection where the language deals with its own semantics. A more formal view of reflection is given by Clavel and Meseguer [1996], essentially defining reflective languages as ones in which there is a mapping between certain data types in the language and a portion of the language’s semantics.

Reflection is a common feature of object-oriented languages, such as Java [Kirby et al. 1998]. In such contexts reflective operations are often coupled tightly to the objects under consideration; for example, one may use reflection to get the value of a field by a string representation of its name, resulting in another object to be casted or reflected again. This differs from our notion of reflection, in which a tree composed of various nonterminals is reflected as a whole into a distinct AST representation, upon which further operations may be performed. In our experience this approach is a better fit for language meta-programming, as any operation on trees is primarily either "structural" (best implemented by attributes on AST) or "semantic" (best implemented by attributes on the nonterminals in question.)

Numerous problems related to performing generic operations on algebraic/inductive datatypes, including substitution and serialization, are described in the scrap-your-boilerplate papers by Lämmel and Jones [2003, 2004]. This is mostly achieved through advanced use of the type system (e.g. type classes), but an approach based on reflection to a generic DataType representation is also described; this is similar to our AST, (representing ADT values, lists, constants etc.) however operations on DataType are expressed using recursive functions and type classes rather than attributes.

The time savings of writing object-language concrete syntax in a meta-language with quote and antique operators has been noted previously and seen in tools such as ASF+SDF [Brand et al. 2001] and Stratego [Visser 2001, 2002]. The Stratego approach is particularly relevant; it differs from our approach in that the transformation from embedded object-language abstract syntax to meta-language abstract syntax happens incrementally through term rewriting, during which ill-sorted intermediate trees composed of meta- and object-language abstract syntax exist. Thus, ill-sorted trees can still be represented. The TypeSmart feature in Stratego [Erdweg et al. 2014] can be used to dynamically disallow ill-sorted trees; when a constructor is applied it checks that the arguments are of the required sort.

In our reflection-based approach, the dynamic checking is only needed for antique productions in the reify operation, the rest of the translation process is checked statically. This comes at the cost of the added complexity of a separate AST representation, not required in a rewriting approach. Instead of reify dynamically type-checking the AST, we could have the AST constructors check the types of their represented argument values, as done with the TypeSmart feature of Stratego. However this would require a way to access (in Silver specifications, not just in its runtime as reify does) the types of productions by name.

Squid quasiquotes [Parreaux et al. 2017a,b] provide concrete object-language syntax embedding for syntax tree construction in Scala. These bear some resemblance to our AST-based approach, most notably by allowing introspection via pattern matching of quoted Code values, but Squid provides stronger type safety guarantees about quoted fragments.

A definition of staged programming and the MetaML language is given by Sheard [1998]. Calcagno et al. [2003] introduces the MetaOCaml language and describes an approach to implementing staged languages by translation to a target language containing constructs for reflection and abstract syntax tree manipulation. This differs from our design, in which a meta-language supplying such constructs is used to build a direct interpreter for evaluating a staged language.

10 Discussion and Conclusion

This paper integrates a form of reflection into attribute grammars with a reflect construct of turning well-sorted trees into a generic representation and an inverse reify operation. In meta-programming systems, such as Silver, in which the type system of the the specification language enforces the well-sortedness of syntax trees, writing analyses and transformations of such trees can involve a lot of boilerplate specifications. We show how reflection lets the language developer reflect a tree into a generic form, process or analyze it in a convenient generic way, and then, in some applications, reify the tree back to the well-sorted form. In the example applications, many lines of boilerplate Silver specification were eliminated from existing applications or were avoided from the beginning. In our experience, even though well-sortedness is not guaranteed by the type system in the AST form we rarely found this to be problem; it is easily outweighed by the savings in lines of specifications written.

There are many additional uses of reflection in attribute grammars beyond those discussed here. Examples include writing visitor-pattern-style traversals over trees, mechanisms for (runtime) type-safe casts, and implementation of generic map and reduce operations over arbitrary trees. Reflection also opens up the possibility building a term-rewriting extension to Silver, something that is a topic of future work. Other future work includes improving the performance of the processing of Silver interface files by replacing the text-based serialization system with one that generates more compact binary representations.
References


